SPECIFICATION

METHOD AND MECHANISM FOR USING A META-LANGAGE TO DEFINE AND ANALYZE TRACES

BACKGROUND AND SUMMARY

Tracing is an approach for logging the state of computer applications at different points during its course of execution. Tracing is normally implemented by inserting statements in the computer application code that outputs status/state messages ("traces") as the statements are encountered during the execution of the code. Statements to generate traces are purposely placed in the computer application code to generate traces corresponding to activities of interest performed by specific sections of the code. The generated trace messages can be collected and stored during the execution of the application to form a trace log.

Programmers often use tracing and trace logs to diagnose problems or errors that arise during the execution of a computer application. When such a problem or error is encountered, trace logs are analyzed to correlate trace messages with the application code to determine the sequence, origin, and effects of different events in the systems and how they impact each other. This process allows analysis/diagnoses of unexpected behavior or programming errors that cause problems in the application code.

In a parallel or distributed environment, there are potentially a number of distributed network nodes, with each node running a number of distinct execution entities such as threads, tasks or processes (hereinafter referred to as "threads"). In many modern computer applications, these threads perform complex interactions with each other, even across the network to threads on other nodes. Often, each of the distributed nodes maintains a separate log file to store traces for their respective threads. Each distributed node may also maintain multiple trace logs corresponding to separate threads on that node.

Diagnosing problems using multiple trace logs often involves a manual process of repeatedly inspecting different sets of the trace logs in various orders to map the sequence and execution of events in the application code. This manual process attempts to correlate events in the system(s) with the application code to construct likely execution scenarios that identify root causes of actual or potential execution problems. Even in a modestly distributed system of a few nodes, this manual process comprises a significantly complex task, very much limited by the capacity of a human mind to comprehend and concurrently analyze many event scenarios across multiple threads on multiple nodes. Therefore, analyzing traces to diagnose applications in parallel and/or distributed systems is often a time consuming and difficult exercise fraught with the potential for human limitations to render the diagnoses process unsuccessful. In many cases, the complexity of manual trace analysis causes the programmer to overlook or misdiagnose the real significance of events captured in the trace logs. With the increasing proliferation of more powerful computer systems capable of greater execution loads across more nodes, the scope of this problem can only increase.

An improved approach to diagnosing computer systems and applications uses trace messages that are materialized in a markup language syntax. Hyperlinks can be placed in the trace messages to facilitate navigation between sets of related traces. One method to generate trace messages having markup language syntax is to first generate trace strings from an application having a known set of fixed formats, in which the process for extracting information to create a new version of the trace in a markup language syntax is driven by knowledge of the position and existence of specific data in the trace strings. This type of approach is described in more detail in co-pending U.S. Patent App. Ser. No. ________, Attorney Dkt. No. 254/254, entitled "Method and Mechanism for Diagnosing Computer Applications Using Traces," filed on even date herewith, which is hereby incorporated by reference in its entirety.

Trace tools that access fixed format traces expect information in the trace string to appear in a predetermined sequence. However, information in the trace string may not be properly recognized if deviations occur from the exact requirements of the fixed format for the trace. With the fixed format trace approach, changes to the trace string format may require changes in the corresponding tools used to parse and tokenize the trace strings, and these changes could involve significant modification or rewrites to the underlying programming code for the trace tools. Yet it may be highly desirable to allow customization of trace string formats without requiring the burden of modifying or rewriting corresponding trace tools.

The present invention provides a method and mechanism for utilizing a meta-language to define and analyze traces. According to an embodiment, non-fixed format traces are used to generate and materialize traces that incorporate markup language syntax. With this aspect of the invention, changes to a trace format do not necessitate code changes in the corresponding tools for navigating through traces. Further aspects, objects, and advantages of the invention are described below in the detailed description, drawings, and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are included to provide a further understanding of the invention and, together with the Detailed Description, serve to explain the principles of the invention.

Fig. 1 shows an example of a communications operation between two network nodes and corresponding trace logs.

Fig. 2 shows trace logs including traces in markup language pseudocode according to an embodiment of the invention.

Figs. 3 and 4 are diagram of system architectures with which the present invention may be implemented.

Fig. 5 is a diagram of a system for utilizing traces which do not need to be in a single fixed format according to an embodiment of the invention.

Fig. 6 is a diagram of hash tables to store results of trace paring operation.

Fig. 7 is a diagram of an example semantic network.

Fig. 8 is a flowchart of a process for utilizing traces according to an embodiment of the invention.

DETAILED DESCRIPTION

The present invention is disclosed in an embodiment as a method and mechanism for implementing tracing and trace logs. The disclosed embodiment of the invention is directed to trace logs for distributed and parallel systems. However, the principles presented here are equally applicable to trace log(s) in other system architecture configurations, including single node configurations, and thus the scope of the invention is not to be limited to the exact embodiment shown herein.

An aspect of one embodiment of the present invention is directed to traces comprising markup language syntax. A markup language is a collected set of syntax definitions that describes the structure and format of a document page. A widely used markup language is the Standard Generalized Markup language ("SGML"). A common variant of SGML is the HyperText Markup Language ("HTML"), which is a specific application of SGML used for the world wide web. The Extensible Markup Language ("XML") is another variant of SGML. For explanatory purposes only, the invention is described using HTML-compliant markup language syntax. However, it is noted that the present invention is not limited to any specific markup language syntax, but is configurable to work with many markup languages.

Analysis of traces is greatly facilitated pursuant to an embodiment of the present invention by using traces implemented with markup language syntax. To illustrate this aspect of the invention, consider a simple communications operation that is performed between two network nodes. Fig. 1 shows Nodes 1 and 2 executing an operation that consists of a first message that is sent from Node 1 to Node 2 and a response message that is subsequently sent from Node 2 to Node 1. Assume that a first trace log 100 maintains traces for threads executing on Node 1 and a second trace log 102 maintains traces for threads executing on Node 2.

When analyzing trace logs for communications operations that send messages between network nodes, it is common for sets of related traces to appear in multiple trace logs across the network. For example, a "send" operation trace in a first trace log at a first node often has a counterpart "receive" operation trace located in a second trace log at a second node. Thus in the example of Fig. 1, the trace for the initial "send" operation from Node 1 to Node 2 is stored in log file 100 as trace message 5000. The trace for the "receive" operation that occurs on Node 2 is stored in trace log 102 as trace message 10000. The trace for the subsequent "send response" operation from Node 2 to Node 1 is stored as trace message 10200 in trace log 102. The trace for the "receive" operation that occurs at Node 1 for the response message is stored as trace message 22000 in trace log 100.

Consider if it is desired to analyze/diagnose this communications operation between Node 1 and Node 2. When a programmer analyzes the set of traces corresponding to that communications operation, it is likely that the programmer must review both the send and receive traces. In this example, the send and receive traces for the communications operation are spread across multiple trace logs on multiple nodes, and the traces of interest may be buried among hundreds or thousands of irrelevant traces that correspond to applications/operations of no immediate interest. Even in this very simple example, analysis of the trace logs could involve a complex and time-consuming task just to identify the traces of interest. That difficult task is compounded by the additional burden needed to manually jump between the different trace logs to chase the chain of traces across the multiple network nodes. In the real world, this analysis/diagnosis task could become far more difficult because of messaging operations that involve many more threads across many more network nodes.

To address this problem, one embodiment of the present invention materializes trace messages using a markup language syntax. By implementing trace messages using markup language syntax, navigational intelligence can be embedded into the trace messages using "hyperlinks." A hyperlink is an element in an electronic document or object that links to another place in same document/object or to an entirely different document/object. As noted above, when a programmer analyzes the set of traces corresponding to that communications operation, it is likely that the programmer must review both the send and receive traces. For this reason, it is useful to link related communications traces at the senders and receivers of inter-nodal messages. Thus, a send trace is hyperlinked to its counterpart receive trace. The hyperlinks can be defined in both the forward and reverse directions. A chain of linked traces can be established whereby each trace is hyperlinked in sequential order to both its predecessor and successor trace. All traces relating to a common operation or activity are therefore linked together via a chain of hyperlinks extending from a first trace through all other related traces.

Fig. 2 depicts trace logs 200 and 202 that include traces messages implemented using markup language pseudocode. Trace log 200 corresponds to trace log 100 of Fig. 1 and contains traces generated by threads on Node 1. Trace log 202 corresponds to trace log 102 of Fig. 1 and contains traces for threads on Node 2. Each of the trace messages for the communications operation shown in Fig. 1 are represented in Fig. 2 using markup language pseudocode to illustrate hyperlinks between related trace messages. In particular, the "send" trace message 5000 in trace log 200 includes a forward hyperlink to its corresponding "receive" trace message 10000 in trace log 202. The "receive" trace message 10200 includes a reverse hyperlink to the "send" trace message 5000. The "send response" trace message 10200 in trace log 202 is forward linked to its corresponding "receive response" trace message 22000 in trace log 200. The "receive response" trace message 22000 is reverse hyperlinked to the "send response" trace message 10200. While not a send and receive pair, trace messages 10000 and 10200 could be hyperlinked together to indicate the sequential nature of the operations corresponding to these traces. Any suitable markup language syntax may be employed to implement this type of hyperlinking.

Once the trace messages have been materialized into a markup language syntax, any browser or viewer capable of interpreting the chosen markup language may be used to navigate the trace log(s). The traces for any activity of interest can be navigated by identifying one of the activity's traces and traversing the chain of hyperlinks extending from that trace – without requiring any manual searching for related traces. Since both forward and reverse hyperlinks can be embedded into the trace log, the traces for an activity of interest can be traversed in both the forward or reverse directions.

One embodiment of the present invention provides a method and mechanism for utilizing non-fixed format traces to generate and materialize traces that incorporate markup language syntax. With this aspect, changes to a trace format do not necessitate code changes in the corresponding tools for navigating through traces.

According to an embodiment of the invention, traces in the system do not have to be in a single fixed format, but each set of traces should correspond to a defined trace format grammars ("TFG"). A trace format grammar is the set of formatting guidelines that defines the placement of information in its associated trace strings. A set of trace format grammars {TFG1, TFG2,..., TFG*n*} is defined for the tracing activities in the computing system, where TFG1 refers to a first defined trace format grammar, TFG2 refers to a second defined trace format grammar, and TFG*n* refers to an *n*th trace format grammar. Potentially, an infinite number of trace format grammars may be defined with each trace format grammar defined to address a different trace string format used in the system. Each time a new trace string format needs to be employed, a new trace grammar format is defined that is suitable for the new trace string format.

Each trace format grammar in the system should comply with a recognized meta-language grammar ("G"). A meta-language grammar ("G") is specified for the system, which comprises a set of rules from which the TFGs are specified. In effect, each trace format grammar uses a unique combination of grammar syntax specified in the meta-language grammar G to form an individual TFG. While each individual TFG include formatting differences from another individual TFG, all TFGs comply with the guidelines set forth in the meta-language grammar G. It is noted that the invention is not limited to the specific meta-language grammar shown herein; multiple different meta-language grammars may be defined and used, with the specific characteristics of the meta-language grammar(s) actually used depending upon the particular application to which the invention is directed. In one embodiment of the invention, each implementation includes only one meta-language grammar.

Fig. 5 shows components of a system 500 for utilizing non-fixed format traces from trace files 508 according to an embodiment of the invention. The system 500 includes a generator mechanism 504 that receives a set of TFGs 502 as input. The set of TFGs 502 comprise the formatting rules used to create each trace in trace files 508. Each TFG in the set of TFGs 502 comprises formatting definitions for a separate trace string format utilized in the trace files 508. The generator 504 verifies that each TFG in the set of TFGs 502 complies with the guidelines set forth in the meta-language grammar G. In addition, the generator 504 verifies that no conflict exists among the set of TFGs 502. If a conflict is identified, the generator 504 may provide an alert regarding the conflict or may optionally perform other operations such as dynamically correcting the error in the TFG and/or trace files 508 or to ignore the conflict. An example of such a conflict is if the generator 504 identifies two separate TFGs that define conflicting formats for trace strings incorporating the same keyword.

The generator 504 thereafter generates and/or passes trace analysis data to a unified trace analyzer mechanism 506 for all TFGs 502. In an embodiment, the analyzer 506 comprises a parser that parses and tokenizes each trace in trace files 508. To accomplish this parsing function, the analyzer 506 includes a set of rules compiled from all the rules used for each TFG in the set of TFGs 502.

As an illustrative example, consider the following grammar (BNF) for a meta-language G, in which the all-upper case letters are keywords of G, and “start” is the start symbol:

```
start :
     KEYWORDS ':' string_list ';'
    STRING ':' string_or_number ';'
    MULTI_RELATION ':' relation_list ';'
    SELF_RELATION ':'  string_list ';'
    RULES ':' rules_of_g ';'
     ;
relation_list :
      one_relation
     | relation_list ',' one_relation
     ;
one_relation :
      STRING "->" string_list;
string_or_number:
     STRING
    | NUMBER
    ;
string_list :
     STRING
    | string_list STRING
    ;
rules_of_g :
     one_rule
    | rules_of_g '|' one_rule
    ;
one_rule :
    key_word UID string_list '{' actions_list '}'
    ;
```

```
key_word : STRING;

actions_list :
        one_action
       | actions_list ‘;’ one_action
       ;
one_action:
        JUMP
       | QUERY query_list ‘;’
       ;
query_list:
        one_query
       | query_list ‘,’ one_query
       ;
one_query:
       query ‘?’ callback_for_query
      ;

query: string_list;

callback_for_query: STRING;
```

The following is an example of a TFG for this meta-language grammar, where keywords of G have been given meaningful names:

```
KEYWORDS : SEND RECV RESOURCE WAIT POST EVENT ;
RULES : SEND UID number address number { JUMP;}
    | RECV UID number address number { JUMP;}
    | RESOURCE UID { QUERY who is my master?callback1, who is blocking
me?callback2; JUMP;}
    | WAIT UID  {JUMP;}
    | POST UID  {JUMP;}
    | EVENT UID {JUMP;}
    ;
MULTI_RELATION: WAIT -> POST EVENT, SEND -> RECV ;
number : NUMBER
    ;
address : NUMBER
    ;
```

This TFG describes the trace string format for a set of traces in trace files 508. Potentially a plurality of such TFGs may be defined, with each TFG compliant with the meta-language grammar G.

In an embodiment, the system 500 internally generates following trace description language that is used as the set of rules to parse traces from trace files 508:

```
rule1 : SEND string number address number { register_jump_routine();};
rule2 : RECV string number address number { register_jump_routine();};
rule3 : RESOURCE string { register_queries_callbacks_and_jump_routines();} ;
rule4 : WAIT UID {register_jump_routine();}
rule5 : POST UID {register_jump_routine();}
rule6 : EVENT UID {register_jump_routine();}
string: STRING ;
number: NUMBER ;
address : NUMBER ;
```

Each of these rules is used by the analyzer 506 to parse traces from trace files 508. Each rule sets forth an expected string format for a trace having a given keyword (such as keywords "send" and "recv"). In one embodiment, these rules are compiled from all the TFGs 402 in the system.

In this example, the actions in the braces are generated to process the arguments for each rule. Thus, a trace having the keyword "SEND" is parsed by the analyzer according to rule1, with the "register_jump_routine()" routine called as a result of recognizing that rule1 applies to this trace statement. Each query can have a callback associated with it which is called whenever a query is selected from the list of queries associated with a rule. In the example, rule3 can instantiate an occurrence of RESOURCE definition in the trace file. Each of these resource occurrences can have queries like "who is owner of this resource ? Get_owner", "who is the master of this resource ? Get_master" etc. From the interface to the system, the user selects one of the occurrences and selects a query, and thereafter the corresponding callback executes and outputs the result to the user. This callback mechanism provides the flexibility to associate algorithms with a query which can be changed without having to change the search engine (e.g., Network builder and Navigator).
As set forth in more detail below, these register routines also help to build the database that tracks the navigational relationships between traces.

The analyzer 506 reviews each trace in the trace files 508 to verify its format and to create a set of data having relevant information needed to create new versions of the traces having markup language syntax. An example of such relevant information includes information that identify traces having particular keywords, such as "send" and "recv" keywords. According to an embodiment, tables are created to store this relevant information. As shown in Fig. 6, the Analyzer 506 creates a separate hash table 602, 604, and 606 for each type of keyword defined in the set of TFGs 502. In the example of Fig. 6, hash table 602 corresponds to the "send" keyword, hash table 604 to the "recv" keyword, and hash table 606 to the "resource" keyword. Each trace on a single node should correspond to an unique trace identifier, with the entries in the hash tables hashed using the unique trace identifier as the key. Each hash table entry contains the relevant information extracted from the trace text file.

Referring back to Fig. 5, a trace miner mechanism 510 utilizes the information compiled by analyzer 506 to build a relational network that represents relationship patterns, e.g., navigational patterns, in the trace files 508. The task of data mining often involves determinations of relationships in data. The trace miner 510 accesses the hash tables produced by the analyzer 506 and also the trace files 508 as input to attempt to find relevant relationships in the trace files 508.

An embodiment of the present invention employs semantic networking to represent relationships identified in trace files 508. A semantic network comprises a structure consisting of nodes and links. In the present invention, nodes represent resources such as locks, semaphores, source of a message, etc. The links define relationships between nodes, such as navigational patterns. Examples of such relationships include "mastered by" "blocked by", and "send to" operations for resources. Many languages and system are available for representing a semantic network, such as SNePS (for which additional information is available at http://www.cse.buffalo.edu/sneps/). Semantic networks may also be represented using SGML or its variants such as XML and HTML.

Fig. 7 shows an example of a semantic network in which node 702 corresponds to a first trace statement and node 704 corresponds to a second trace statement. In this example, the trace statement of node 702 relates to a "send" operation that is captured at line 1162 of a first trace file "trace1.txt." The trace statement of node 704 relates to a corresponding "recv" operation that is captured at line 2211 of a second trace file "trace2.txt." Note that both nodes share a common unique identifier "23ae3". This is the unique identifier that permits ready identification of a relationship between the two underlying trace statements for these nodes, and consequently indicates that a link 706 should extend between nodes 702 and 704. In this example, since the nodes represents a send-recv pair, the link 706 represents a "send to" relationship between the two nodes 702 and 704.

Referring back to Fig. 5, a network builder mechanism, shown as the Diagnostic Navigational Pattern Builder or "DNP" Builder 514, performs the function of constructing one or more disjoint semantic networks representing relationships between various traces in trace files 508. The DNP Builder 514 uses the relationships among the keywords as described in the TFGs to discover identifiable patterns and relationships. The DNP Builder 514 uses the hash tables and the keyword relationships created by the Analyzer and attempts to create relationship networks. Sometimes relationships may exist between two entries in the same hash table. Relationships may also exist between different hash tables. The DNP Builder 514 may employ any standard mechanism to represent these relationships (e.g., SGML, XML, SNePS or any other language or system capable of representing a semantic network.

These semantic networks may be persistently stored in a database 518 and can be retrieved when required. According to an embodiment, the DNP Builder 514 needs to be run only once for each set of trace files and of TFGs.

A network navigator mechanism, shown as DNP Navigator 516, retrieves the networks from the database 518 and performs the function of searching the networks for particular resources or relationships between resources. Instructions may be provided to the DNP Navigator 516 to retrieve particular nodes or a sub-network corresponding to a given relationship. For example, a user may be interested in knowing the master owner of a particular lock resource. The DNP Navigator 516 may be instructed to search through the set of networks and attempt to match the node and relationships to retrieve all the nodes or sub-networks matching the query.

The DNP Navigator 516 supplies a manifestation tool 512 with trace network data corresponding to traces to be displayed to a user. The manifestation tool 512 is a user interface or browser component and could be constructed using a programming language such as Java. The manifestation tool 512 acts as a renderer for semantic networks built by the DNP Builder 514. The manifestation tool 512 inputs sets of networks from the trace miner 510 along with raw trace statements from trace files 508 and creates appropriately modified versions of the trace files using markup language syntax.

Fig. 8 depicts a flowchart of an embodiment of this aspect of the invention. At 802, a query is received at the manifestation tool 512 pertaining to one or more resources. The manifestation tool 512 instructs the DNP Navigator 516 to search database 518 for any networks corresponding to the queries resources that have previously been constructed by DNP Builder 514 (804). If such networks are found, then those networks are returned to the manifestation tool 512 (806). These networks identify trace statements having interrelationships that are candidates for including navigable hyperlinks. The manifestation tool 512 also retrieves traces from trace files 508 corresponding to the query (808). Based upon the network data retrieved from the DNP Navigator 516, the manifestation tool 512 creates modified versions of some or all the traces to include hyperlinks to related traces (810). These modified versions of the traces are thereafter materialized for viewing by a browser or viewer.

To illustrate the invention, consider if a user would like to view a navigable version of the trace statements corresponding to nodes 702 and 704 in Fig. 7. The manifestation tool 512 would (a) instruct DNP Navigator 516 to search for and retrieve the semantic network shown in Fig. 7, (b) identify the source trace statements corresponding to this semantic network (e.g., line 1162 from trace1.txt and line 2211 from trace2.txt), (c) retrieve these trace statements from trace files 508, (d) create a modified version of these trace statements to include navigable hyperlinks corresponding to link 706, and (d) materialize the modified versions of these traces in a markup language syntax usable by the user's browser or viewer.

Any query may be posed to the manifestation tool 512 regarding one or more resources. Callbacks, such as user-defined callbacks in a TFG, may be specified as part of a query. The manifestation tool 512 or callback would pass the query to the DNP Navigator 516. The DNP Navigator 516 searches through the sets of networks and return the appropriate nodes to manifestation tool 512. Both the manifestation tool 512 and/or the callback should able to communication with the DNP Navigator 516 because a query may require complex algorithmic interpretations of traces and may be possible only if user defined callbacks are specified.

SYSTEM ARCHITECTURE OVERVIEW

Referring to Fig. 3, in an embodiment, a computer system 320 includes a host computer 322 connected to a plurality of individual user stations 324. In an embodiment, the user stations 324 each comprise suitable data terminals, for example, but not limited to, e.g., personal computers, portable laptop computers, or personal data assistants ("PDAs"), which can store and independently run one or more applications, i.e., programs. For purposes of illustration, some of the user stations 324 are connected to the host computer 322 via a local area network ("LAN") 326. Other user stations 324 are remotely connected to the host computer 322 via a public telephone switched network ("PSTN") 328 and/or a wireless network 330.

In an embodiment, the host computer 322 operates in conjunction with a data storage system 331, wherein the data storage system 331 contains a database 332 that is readily accessible by the host computer 322. Note that a multiple tier architecture can be employed to connect user stations 324 to a database 332, utilizing for example, a middle application tier (not shown). In alternative embodiments, the database 332 may be resident on the host computer, stored, e.g., in the host computer's ROM, PROM, EPROM, or any other memory chip, and/or its hard disk. In yet alternative embodiments, the database 332 may be read by the host computer 322 from one or more floppy disks, flexible disks, magnetic tapes, any other magnetic medium, CD-ROMs, any other optical medium, punchcards, papertape, or any other physical medium with patterns of holes, or any other medium from which a computer can read. In an alternative embodiment, the host computer 322 can access two or more databases 332, stored in a variety of mediums, as previously discussed.

Referring to Fig. 4, in an embodiment, each user station 324 and the host computer 322, each referred to generally as a processing unit, embodies a general architecture 405. A processing unit includes a bus 406 or other communication mechanism for communicating instructions, messages and data, collectively, information, and one or more processors 407 coupled with the bus 406 for processing information. A processing unit also includes a main memory 408, such as a random access memory (RAM) or other dynamic storage device, coupled to the bus 406 for storing dynamic data and instructions to be executed by the processor(s) 407. The main memory 408 also may be used for storing temporary data, i.e., variables, or other intermediate information during execution of instructions by the processor(s) 407. A processing unit may further include a read only memory (ROM) 409 or other static storage device coupled to the bus 406 for storing static data and instructions for the processor(s) 407. A storage device 410, such as a magnetic disk or optical disk, may also be provided and coupled to the bus 406 for storing data and instructions for the processor(s) 407.

A processing unit may be coupled via the bus 406 to a display device 411, such as, but not limited to, a cathode ray tube (CRT), for displaying information to a user. An input device 412, including alphanumeric and other columns, is coupled to the bus 406 for communicating information and command selections to the processor(s) 407. Another type of user input device may include a cursor control 413, such as, but not limited to, a mouse, a trackball, a fingerpad, or cursor direction columns, for communicating direction information and command selections to the processor(s) 407 and for controlling cursor movement on the display 411.

According to one embodiment of the invention, the individual processing units perform specific operations by their respective processor(s) 407 executing one or more sequences of one or more instructions contained in the main memory 408. Such instructions may be read into the main memory 408 from another computer-usable medium, such as the ROM 409 or the storage device 410. Execution of the sequences of instructions contained in the main memory 408 causes the processor(s) 407 to perform the processes described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and/or software.

The term "computer-usable medium," as used herein, refers to any medium that provides information or is usable by the processor(s) 407. Such a medium may take many forms, including, but not limited to, non-volatile, volatile and transmission media. Non-volatile media, i.e., media that can retain information in the absence of power, includes the ROM 409. Volatile media, i.e., media that can not retain information in the absence of power, includes the main memory 408. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise the bus 406. Transmission media can also take the form of carrier waves; i.e., electromagnetic waves that can be modulated, as in frequency, amplitude or phase, to transmit information signals. Additionally, transmission media can take the form of acoustic or light waves, such as those generated during radio wave and infrared data communications.

Common forms of computer-usable media include, for example: a floppy disk, flexible disk, hard disk, magnetic tape, any other magnetic medium, CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, RAM, ROM, PROM (i.e., programmable read only memory), EPROM (i.e., erasable programmable read only memory), including FLASH-EPROM, any other memory chip or cartridge, carrier waves, or any other medium from which a processor 407 can retrieve information. Various forms of computer-usable media may be involved in providing one or more sequences of one or more instructions to the processor(s) 407 for execution. The instructions received by the main memory 408 may optionally be stored on the storage device 410, either before or after their execution by the processor(s) 407.

Each processing unit may also include a communication interface 414 coupled to the bus 406. The communication interface 414 provides two-way communication between the respective user stations 424 and the host computer 422. The communication interface 414 of a respective processing unit transmits and receives electrical, electromagnetic or optical signals that include data streams representing various types of information, including instructions, messages and data. A communication link 415 links a respective user station 424 and a host computer 422. The communication link 415 may be a LAN 326, in which case the communication interface 414 may be a LAN card. Alternatively, the communication link 415 may be a PSTN 328, in which case the communication interface

414 may be an integrated services digital network (ISDN) card or a modem. Also, as a further alternative, the communication link 415 may be a wireless network 330. A processing unit may transmit and receive messages, data, and instructions, including program, i.e., application, code, through its respective communication link 415 and communication interface 414. Received program code may be executed by the respective processor(s) 407 as it is received, and/or stored in the storage device 410, or other associated non-volatile media, for later execution. In this manner, a processing unit may receive messages, data and/or program code in the form of a carrier wave.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. For example, the reader is to understand that the specific ordering and combination of process actions shown in the process flow diagrams described herein is merely illustrative, and the invention can be performed using different or additional process actions, or a different combination or ordering of process actions. The specification and drawings are, accordingly, to be regarded in an illustrative rather than restrictive sense.